File management on the grid

Version 0.13

David Adams

January 5, 2005

Introduction

Computing grids offer individual users the opportunity to harness the computing power of thousands of compute nodes and, consequently, the possibility to process and produce vast quantities of data. Most of this data will be stored in files, and the primary task of a grid FMS (file management system) is to ensure that the data output from one operation is available to serve as input for subsequent operations. We draw on experiences gained in the DIAL [1] and ADA (ATLAS distributed analysis) [2] projects to flesh out requirements and develop an interface for an FMS.

DIAL and ADA

DIAL and ADA aim to insulate users from direct interaction with files. A user defines a job by specifying an input dataset and a transformation, and then an analysis service carries out the transformation on the input dataset to produce an output dataset. A dataset may directly include a list of logical files or may contain information that can be used to construct such a list. These are typically logical files, rather than physical file names, so that access is not restricted to compute nodes that have mounted a particular file system. For any given logical file, there may be multiple physical file replicas.

Datasets allow us to separate the problem of selecting the data of interest from that of managing files and replicas. The former is handled by assigning metadata to the datasets and providing means to make queries based on this information and, as such, is outside the realm of the FMS. We do recognize the need for file-level metadata, such as file size, checksum and creation time, and we expect the FMS to maintain such information; however, these values can be evaluated internally and are not part of the user interface.

Other users

Despite these ambitions, users and other processing systems will come into contact with files before and when files are entered into the ADA/DIAL system, e.g. from the detector or an external production or reconstruction system, and when and after they are extracted from the system, e.g. when examining the final ntuples or histograms. In addition the FMS should support users who wish to work entirely outside the ADA/DIAL system, e.g. in the production of small private data samples. Thus, the interface presented here is intended to meet the needs of the internal DIAL/ADA processing system, users of that system, and users producing and analyzing data outside that system.

FMS

An FMS is responsible for cataloging logical files, physical replicas, and their associations and for moving data (creating new replicas). It is also responsible for managing these files, i.e. maintaining lifetime information so the obsolete files can be deleted to reduce clutter and recover storage space. These capabilities are described in the following sections.

There are many systems which provide some of these capabilities. A partial list relevant to ATLAS includes AFS [3], Castor [4], HPSS [5], Magda [6], RLS [7], SRM [8] and dCache [9]. Within ATLAS, Don Quijote (aka DQ) [10] is an attempt to tie some of these pieces together. DIAL defines a C++ interface FileCatalog through which the analysis service accesses an FMS. Implementations based on AFS and Magda are in use today.

Clients and use cases

Who are the clients of the FMS? In a typical scenario, a job or user has a physical file to insert into the FMS. Later, a user or processing system will identify computing resources, copy the files to a nearby location and then ask to make the files accessible for processing with those resources. An alternative processing strategy which avoids the copy is to find the data and send the processing jobs to nearby compute nodes.

Thus we identify five use cases:

1. Put: A client with a physical file inserts into the FMS and gets back a logical file.
2. Copy: A client requests data be moved to a particular location.
3. Stage: A client requests that a replica of a logical file be made readily available.
4. Get: A client user uses the FMS to gain access to a physical replica.
5. Locate: A client asks for the locations of the replicas of a given logical file.

Logical and physical file names

In the simplest logical file model, there are logical file names (LFN's) and physical file names (PFN's). The LFN is a string guaranteed unique in the context (namespace) of a replica catalog and the PFN is a posix file name. However, this model turns out to be too restrictive. Some applications can handle input and even output of non-posix files, e.g. Castor's rfio or dCache's dcap protocols. SRM introduces intermediate names: site and transfer URL's (SURL and TURL).

A file name may be logical in one scope and physical in another. For example, a SURL might be a physical name in an ATLAS replica catalog but act as a logical name when interacting with an SRM service. Similarly, the dCache identifier returned in a get operation from the ATLAS file catalog or an SRM service serves as the input for the get operation from a dCache service.

In this spirit, we will often speak of file references rather than distinguish between logical files, physical files, SURL's and TURL's. These references include a context and a unique identifier within that context. For each context, we assign one or more unique protocol names. We adopt standard [11] (or at least common) URL naming for the logical files using the protocol name as the URL scheme. Table 1 gives some examples.

Context	protocol	Example URL	Path
Replica catalog	guid	guid:1234-5678	
Replica catalog	lfn	lfn://atlas/dc2/runs.dat	
SRM server	srm	srm://srm.bnl.gov/atlas/dc2/runs.dat	
Grid FTP	gsiftp	gsiftp://atlasse.iu.edu/dc2/runs.dat	
FTP	ftp	ftp://atlas.cern.ch/public/members.dat	
Web	http	http://www.atlas.org/members.dat	
DCache server	dcap	dcap://dcache.nmu.edu/atlas/dc2/runs.dat	/pnfs/…
Castor	rfio	rfio://castor.cern.ch/atlas/dc2/runs.dat	/castor/…
AFS	afs	afs://usatlas.bnl.gov/atlas/dc2/runs.dat	/afs/…
NFS server	nfs	nfs://fs01.bnl.gov/home/atlas/dc2/runs.dat	/…
Local file system	file	file://atlas049.ucc.edu/home/atlas/runs.dat	/…

Table 1. Examples of logical file specifications.

In some contexts, files may also be identified with posix file names. For these, the relevant part of the filename path is indicated in table 1. These names are typically not portable, i.e. will not exist on a different computer or worse might even point to an unrelated file.

Virtual organization

The largest scope we consider for file sharing is that a *virtual organization* (*VO*) which may include members and resources from many real organizations (laboratories, universities, corporations, etc.). The VO defines policies by which file identifiers are uniquely assigned. Different VO's may share storage or processing resources and must avoid conflicts, e.g. by using the VO name as the first field in the LFN.

Storage Element

The FMS is responsible for archiving files, i.e. replicating or taking ownership of a physical file, assigning it a handle and then later allowing clients to use this handle to gain access to a replica of the file. The responsibility for archiving is often shared between members a virtual organization (VO) which wishes to present a common view of a large number of such files. In addition, computing resources are geographically dispersed and it is natural that data be cached at the locations where it is (or will be) used to mitigate the effects of finite network bandwidth and latency.

A *storage element* (*SE*) is a localized collection of accessible files with a clear policy for lifetime management. An SE may be used for archiving, caching or both. A VO typically distributes the responsibility for archiving over multiple SE's. Ideally any localized collection of compute resources can be associated with at least one nearby SE to provide caching of input and output data. Most of our use cases can benefit from caching and we assume use of an SE in these cases. To keep the client simple, we assume that a client interacts only with a single SE and this SE interacts with other SE's (and other data sources) to move data in or out. This interaction may be direct (peer-to-peer), mediated by a central management system (e.g. a file transfer service) or some combination of the two.

An SE is expected to provide a unique identifier for each file it manages. These file identifiers are URL's with a common protocol called the native protocol for the SE. A typical SE is SRM-based and thus has srm as its native protocol.

An SE is capable of staging its files, i.e. making them available with a protocol specified by the user, e.g. an SE with srm as its native protocol might be capable of staging files with gsiftp, dcap and nfs protocols. The stage may require creation of another physical copy and so there is a lifetime associated with the staged file distinct from the lifetime of the SE file.

Replica catalogs

All but the first two entries in table 1 are URL's that are naturally associated with a particular location, possible associated with an SE. However, when recording a file reference (e.g. as part of a dataset), it is often desirable to have a location-independent identifier, e.g. to enable a client to retrieve a replica without the need to access the location of the original replica. This is accomplished by means of a replica catalog (RC) which holds the association between a logical file identifier and a collection of replica identifiers. The logical file identifier is typically an LFN or GUID, here one of the corresponding URL's in table 1. The replica identifier is specified using any of the other protocols in table 1.

The use of a replica catalog makes it possible to refer to files with GUID or LFN file identifiers but does not require it. We envision a system where long-lived datasets hold RC URL's (i.e. guid or lfn protocol) but their short-lived counterparts may hold any of the location-specific URL's. Examples of the latter include datasets carrying intermediate or partial results and datasets used to specify jobs after data has been staged. A typical pattern would be to use a GUID or LFN where the replicas may come from any location and to use the location-specific identification to specify the replica be obtained from a particular SE.

File reference scopes

We assume that each VO makes use of a replica catalog (or hierarchy thereof) and that each file managed by the VO has a VO URL which serves as the logical file identifier in that catalog. This file reference is said to be at VO scope. Furthermore, we assume the file replicas are managed by SE's and each replica identifier in the VO RC is a URL in the scope of one of those SE's. An SE may then be asked to stage an SE URL resulting in a staged URL, i.e. a URL at staged scope. Any other URL is said to be external or at external scope. Table 2 lists these scopes and some of the protocols typically associated with each.

Scope	Typical protocols
VO	guid, lfn
SE	srm, gsiftp
Staged	nfs, dcap, rfio, gsiftp
External	file, gsiftp

Table 2. File reference scopes.

A client with no knowledge of data placement is likely to reference a file within the scope of the VO, e.g. with an LFN. After a processing site is chosen, the file is copied (if needed) to a nearby SE and can be referenced in the scope of that SE, i.e. with its native protocol, e.g. srm or gsiftp. Before processing, the SE stages the file and returns a staged URL, e.g. nfs if the requestor is local or gsiftp if remote. Local users might also request other specialized protocols such as rfio or dcap. A remote client will copy the file out of the domain of the FMS into an external scope, typically resulting in the file protocol. A job produces new files, also typically at external scope. A local client might put such a file into the FMS using the file protocol while a remote client could use gsiftp.

Access control

Files hold data that is the endpoint of an intellectual activity and it is natural that the original creator (or some later owner) of a file may want to restrict access to that file. We assume the existence of an access control system where a user desiring access to a file presents credentials, the credentials are mapped to a user identity and the system grants or denies access based on the identity of the user and the file. We do not present an explicit interface but assume the FMS makes use of such a system which. Users identify files with URL's and present credentials consistent with VO policy when they interact with the FMS. The FMS is responsible for enforcing the access control policy. In particular, files must be staged in manner consistent with this policy.

It is likely that early implementations of the FMS will assign each file to a VO and grant read only access to all members of that VO. Authorization will be based on GSI credentials.

Ownership and lifetime management

We consider three models for file lifetime management:

1. Eternal: files are never deleted.
2. Claim: files are claimed kept until the claim is released.
3. Expiration: files are assigned a lifetime which is used to calculate and record an expiration, i.e. the time when the file may be removed from the system.

Space usage grows without bound in the first case and is inevitably exhausted. The "solution" is human intervention or some sort of automatic cleanup. Both end up deleting files that are needed and do not scale well. The second also grows without bound (albeit slower) as failed processes and forgetful users neglect to release claims. In the third model, the number of files is bounded by the product of the request rate and the average lifetime per request but this number may be very large if the lifetimes are not kept to reasonable values.

Our interface assumes a mix of the second and third policies. Whenever a user creates a logical file, copies a file to an SE, or stages a file, a claim is created and assigned an expiration time. As long as the claim exists in the FMS, the claim owner may update its lifetime or release the claim. A file may be removed when all its claims have expired. Claims are removed before a file is deleted and any time after they have expired. The lifetime of a file is defined to be the time until it its longest claim expires. However, the FMS is not obligated to immediately remove the file at the end of this lifetime and it may be possible to resurrect a file by updating the lifetime of one of its claims.

Separate claims are made on VO, SE and staged files. Claims on a VO file are defined in the scope of the VO and claims on SE and staged files are defined in the scope of the corresponding SE. We expect that claims on staged files will typically come from ordinary users or a processing system acting on their behalf and that these claims will be made before or when a processing job starts and released when the job ends. Claims on SE files may be used for archiving or site caching, i.e. so a higher level data management system (DMS) can guarantee that a site maintains a copy of a file indefinitely or some fixed period of time. Claims on a VO file may be held by one acting in the role production manager, analysis group data manager or

ordinary user. The existence of an unexpired claim on such a logical file likely implies that the DMS has a corresponding SE claims on one more replicas.

FMS interface

Here we describe the operations that the FMS must provide to satisfy our use cases. We assume the existence of a VO to define a VO namespace with RC entry points to record and retrieve logical-replica associations. We assume the existence of a collection of SE's to archive the replicas referenced by the VO. We make use of a nearby SE to cache input and output files and as a location to store short-lived files that are not recorded in the VO catalog.

Most operations return a string, typically a URL, an error code and a request ID. In the event of failure, the string is empty and the error code nonzero. The request ID may be used to check the status of operations which return before they are complete. The request ID may also be used to check the status of recently completed operations and possibly to access other information.

For all operations, it is assumed that identity and role of the caller is known.

Many operations require a VO identifier. This is a simple string that should uniquely identify a VO. E.g., for ATLAS, the value might simply be "atlas". Similarly, many operations require an SE identifier. This is also a string typically, the prefix of URL's native to that SE, e.g. "srm://srm-atlas.bnl.gov".

All operations take a blocking timeout which is the maximum time that may be taken before returning control to the caller. Where relevant, operations also take a non-blocking timeout which is the maximum time to take after return to complete the operation. If the latter value is zero, the operation fails if it is not completed before return.

It is expected that clients will establish a connection to the FMS that assigns default values for many of the parameters including VO and SE identifiers, stage, SE and VO lifetimes, blocking and non-blocking timeouts and the naming policy (see below). Most of the arguments listed for each method would typically be specified when this connection is established rather than each time the operation is invoked.

Stores: return the list of SE's

The *stores* operation returns the (space separated) list of known SE's, at least those associated with the VO. The client provides the VO identifier.

Scope: return the scope of a URL

The *scope* operation returns the scope of a URL for a given VO. The client provides the following:

1. VO identifier
2. URL

The returned string is the VO identifier if the URL is in the VO scope, the SE identifier if it is in the scope of an SE in the VO, and "external" otherwise. No attempt is made to discern if this is a staged file.

Locate: return the location of a file

The *locate* operation returns the location of a file. Input parameters are:

1. VO identifier
2. URL

If the input is a VO URL, the returned string is a space-separated list of identifiers for SE's that are holding replicas of that file. If the input URL is in the scope of a known SE, the output is the identifier of that SE.

Put: create a file in the SE and VO

The *put* operation copies a file into the SE and the VO. The client provides the following:

1. VO identifier
2. SE identifier
3. URL for the input file
4. Name
5. Naming policy
6. VO lifetime
7. SE lifetime
8. GUID
9. Blocking timeout
10. Non-blocking timeout

The input URL may not be in the scope of this VO. If the URL is in the scope of this or a foreign (but known) SE, then the URL is registered in the VO RC. If the URL is in the scope of a foreign SE or is external, then the file is copied to this SE and this replica is registered in the VO RC. If provided and relevant, the GUID is registered with the VO.

No VO registration is done if the VO lifetime is zero. The SE lifetime must be nonzero in this case. Users who wish to separate the copying of the file to the SE from registration with the VO may first call *put* with the VO lifetime set to zero, wait for that operation to complete, and then call *put* with the URL returned in the first call. The GUID is set in the second call.

The input URL and name are used in accordance with the naming policy to create the SE and VO URL's.

If successful, the operation returns the VO URL if the input VO lifetime is nonzero and the SE URL otherwise. The caller is granted a claim to the returned file with the specified lifetime. If both lifetimes are nonzero, the caller is additionally granted a claim to the SE file.

This operation must have access to the input file which is frequently a local file. A part of this operation might be implemented on the client side to copy this file to a location accessible to the SE. This operation should delete that intermediate file (or assign its management to the SE) before completion.

Copy: bring a replica to the SE

The *copy* operation requests that the SE obtain a replica of a file. The input parameters are:

1. VO identifier
2. SE identifier
3. URL for the input file
4. SE lifetime
5. Blocking timeout
6. Non-blocking timeout

The input file must exist in the scope of the VO or a known SE. If the URL is at VO scope and there is not yet a replica for this SE, then one is copied over and the VO RC is updated. If the URL is in the scope of a foreign SE, the file is copied to this SE and the VO RC is not updated. In this case, subsequent requests with the same URL may result in another copy. Use *put* if the input file is foreign.

If successful, the output of this operation is an SE URL with the protocol native to the SE. The operation schedules the copy but its success does not guarantee the copy has taken or will take place. Upon successful completion of the copy, the caller is granted a claim to the SE file for at least the requested lifetime.

Stage: staging a replica

The *stage* operation requests that the SE make a local replica available for processing. This might entail copying from another SE, restoring from tape or making a local copy. The input parameters are:

1. VO identifier
2. SE identifier
3. URL for the input file
4. Protocol list
5. Staged lifetime
6. Blocking timeout
7. Non-blocking timeout

Again the input file must exist in the scope of the VO or a known SE. The protocol list is an ordered list of acceptable protocols. The SE will stage the file in accordance with the first acceptable protocol and grant the caller a claim to that staged file for at least the requested lifetime. If needed, the file is first copied and registered as for the *copy* operation.

The output of the operation is the URL of the staged file. Success does not guarantee the file has been or will be staged. The caller is granted a claim to the file for at least the requested lifetime.

Get: accessing a replica

The *get* operation also stages an accessible replica. The arguments are the same as the *stage* operation except the non-blocking timeout is not provided. The effect is the same as calling that operation with non-blocking timeout set to zero, in particular the caller is granted a claim to the returned file.

Extract: extract a replica

The *extract* operation extracts a replica from the FMS.

1. VO identifier
2. SE identifier
3. URL for the input file
4. URL for the output location
5. Blocking timeout

Once again, the input file must exist in the scope of the VO or a known SE. The output URL is external and may be incomplete. The FMS uses it as a hint to construct the actual URL which is returned. The FMS does not manage (i.e. delete) the returned file.

Unlike the *copy*, *stage*, and *get* operations, the FMS may not have write access to the location for the destination file and some of this operation may need to be carried out on the client side. A typical scenario would be to get the file with the gsiftp protocol on the server side and then copy it to a local file.

Has: checking the existence of a file

The *has* operation checks whether a file exists in the scope of the VO, a known SE, or the stage of the default SE. The input arguments are:

1. VO identifier
2. SE identifier
3. URL

If the argument is at VO scope, the operation returns if the URL exists for this VO. If the argument is at SE scope, the operation returns if the corresponding SE has this entry. Otherwise it is assumed the URL is at staged scope and the operation returns if a replica has been staged with the specified URL.

The status associated with this operation includes the complete list of claims associated with the file.

The output URL from a non-RC *put*, a *copy* or a *stage* operation may be used with *has* to check whether that operation has completed successfully.

Lifetime: return the lifetime of a file

The *lifetime* operation has the same arguments and behaves the same as *has* except that it returns the remaining lifetime of (i.e. the lifetime of the longest claim on) the input file. It returns 0 if the file does not exist or there are no unexpired claims. Note that a file will exist with zero lifetime if it has expired but has not yet been removed.

Claimed: check if the caller has claims on a file

The *claimed* operation checks whether the caller has any claims on a file. The arguments are the same as for has. The status associated with this operation holds the list of claims.

Claim: return the description of a claim

The *claim* operation takes a claim ID as input and returns the owner, the owner's role, the file URL and the expiration time for the claim. Input arguments are:

1. VO identifier
2. SE identifier
3. Claim ID

Extend: extend the lifetime of a claim

The extend operation resets the expiration time of a claim to at least the specified lifetime. If successful, the associated file URL is returned. The input arguments are:

1. VO identifier
2. SE identifier
3. Claim ID
4. Lifetime

A lifetime of zero may be used to cause the claim to expire without removing the claim. If the requested lifetime is shorter than the assigned lifetime, then the latter may be decreased.

Release: release a claim on a file

The *release* operation informs the system that the specified claim may be dropped. Input arguments are:

1. VO identifier
2. SE identifier
3. URL

Status

The *status* operation provides means to check the complete status of *put*, *copy*, *stage* and other operations. The input is the request ID returned by the operation. The status indicates whether the operation has completed and if not, an estimated time to completion is returned. If completed, then the status should indicate success or failure. In the case of failure an error code or message should be included. If successful, the ID's of any associated claims are included. It should also be possible to discover all the requests associated with a particular user.

File lists

All the operations taking a single file reference should also be implemented with a list of input files and return a corresponding list of output file references.

Accounting

Our model lends itself nicely to accounting. Claims on files provide natural means to ensure the files stored at the VO, SE and staged scopes are still needed and who (and in which role) has as indicated the need. Costs for data transfer, bookkeeping and space allocation can be charged accordingly. Charges based on this usage could be used to discourage users from making unnecessary requests or requesting excessive lifetimes. Publication of the usage could also serve the same purpose.

Implementation

User interface

DIAL is implemented in C++ and so it is natural to construct an OO implementation in that framework. The existing FileCatalog interface can be replaced with an abstract class FileManager which provides the above operations as member functions. Specific implementations would be constructed as subclasses.

A command line interface is more natural to other users and DIAL transformation scripts. The above operations could be implemented as commands (fmsput, fmsget …). Different implementations could be provided in different software packages. Default parameters could be taken from a configuration file.

Client-service

The FMS is naturally implemented as a collection of services, each making use of a local SE (SRM or gsiftp directory) and global or local VO replica catalog. These services provide the FMS interface, link the local SE and RC, and provide missing functionality such as claim management, lifetime enforcement and staging. Lightweight clients would support remote users leaving the bulk of the work (software installation, configuration, storage management, database administration, *etc*.) to the storage sites. Above we indicated a couple operations (*put* and *extract*) where it may be desirable to provide some client side code.

The interface (WSDL in the likely case we use web services) to this service would provide operations very close to the ones described above. A client implementation would have little work to do except to add some file transfer capabilities the *put* and *extract* operations. The client would also have the responsibility to discover the appropriate service instance based on the user's choice of VO and SE.

It is already a common pattern in DIAL to construct a service based on a class interface and then make an implementation of that interface that is a client to the service.

Existing tools

It is likely that any useful implementations will be based on existing replica catalogs and storage services. For ATLAS, DQ already provides a common interface to the various replica catalogs and is capable of presenting a VO-wide view. The lifetime management of VO files is not addressed.

Almost all of the replica identifiers in the ATLAS catalogs make use of the gsiftp protocol. Again DQ provides access to the associated storage elements. We would like to provide ATLAS users with the possibility to stage other protocols such as file, rfio and dcap and to have better tracking of ownership and lifetime of the SE files. Some of this functionality could come from various SRM implementations including those based on Castor and dCache.

A central question of great practical interest to ATLAS is whether DQ should be extended to provide these functionalities and the interface described here or we should implement what is described here as a layer over DQ and other tools.

Conclusions

An interface for a grid file management system has been outlined including a list of supported operations. The system meets the requirements of the DIAL/ADA system and those of ordinary users. Clients interact with an FMS and can store and later retrieve files at the VO or SE level. The SE enables users to stage

these files in various protocols. A claim-based system allows users and data and workload management systems to control the lifetimes of files in the system.

We deliberately do not address issues such as bulk data management, batch data transfer, automatic data migration and metadata on the assumption that these are not of direct interest to the average user or can be handled via independent interfaces.

References

1. The DIAL home page is http://www.usatlas.bnl.gov/~dladams/dial.
2. The ADA home page is http://www.usatlas.bnl.gov/ADA.
3. One description of AFS is at http://www.openafs.org.
4. The Castor home page is http://castor.web.cern.ch/castor.
5. The HPSS home page is http://www.hpss-collaboration.org/hpss/index.jsp.
6. The Magda home page is http://www.atlasgrid.bnl.gov/magda/info.
7. RLS is described at http://www.isi.edu/~annc/RLS.html.
8. SRM is described at https://forge.gridforum.org/projects/gsm-wg.
9. The dCache home page is http://www.dcache.org.
10. Don Quijote is described at http://mbranco.home.cern.ch/mbranco/cern/donquijote.
11. URL's are discussed at http://www.w3.org/Adressing.